ELSEVIER

Information and Software Technology 43 (2001) 801–812

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

An investigation of coupling, reuse and maintenance in a commercial C++ application

F.G. Wilkie[a,*], B.A. Kitchenham[b]

[a]*Faculty of Informatics, Centre for Software Process Technologies, University of Ulster, Newtownabbey, Co. Antrim BT37 0QB, Northern Ireland, UK*
[b]*Department of Computer Science, Keele University, Keele, Staffordshire ST5 5BG, UK*

Received 4 September 2000; revised 7 September 2001; accepted 7 September 2001

Abstract

This paper describes an investigation into the use of coupling complexity metrics to obtain early indications of various properties of a system of C++ classes. The properties of interest are: (i) the potential reusability of a class and (ii) the likelihood that a class will be affected by maintenance changes made to the overall system. The study indicates that coupling metrics can provide useful indications of both reusable classes and of classes that may have a significant influence on the effort expended during system maintenance and testing. © 2001 Elsevier Science B.V. Open access under CC BY-NC-ND license.

Keywords: Object oriented metrics; Complexity; C++; Coupling; Reuse; Maintenance; Software change

1. Introduction

Coupling is well recognised for its contribution to software design. It can have a significant influence on system integration and maintenance costs. Ideally, interacting objects should be as loosely coupled to one another as possible.

Fyson and Boldyreff [1] suggest that up to 80% of total software lifecycle costs are often consumed during the maintenance phase. Measuring the coupling between classes is important during maintenance in (a) predicting various software process costs such as systems integration; (b) determining where preventative maintenance might be applied to best effect within a system of interacting classes and (c) assessing the impact of changes on the software system — the so-called impact analysis [2].

Measuring couplings between classes provides a way of pre-empting some issues associated with impact analysis. At the stage when the couplings are determined, the precise changes to be made will not usually be known. However, when a change request has been generated, knowledge of the precise couplings between classes can aid the subsequent impact analysis process by highlighting areas of an application that may result in very significant knock-on effects if they have to be modified.

The class couplings measured in this study relate to the aggregation and association relationships. No consideration is given to inheritance relationships. Lindvall [3] has investigated the nature of changes to a commercial C++ system at Ericsson Radio Systems. His findings suggest that "the class and inheritance structure are stable and remain mostly unchanged" during the maintenance phase. This finding is in agreement with our own findings [4] where, for example, the Depth of Inheritance (DIT) and Number of Children (NOC) metrics of Chidamber and Kemerer [5] changed in only two and four classes, respectively, out of the set of 114, for the application studied here, over a 2 1/2 year maintenance period.

Measuring internal system couplings can be achieved in many ways [6]. At present, it is not clear precisely which ways can provide the optimum diagnostic ability. Presumably the most detailed coupling measures will be best, but are any small refinements to the basic Chidamber and Kemerer [5] coupling metric, CBO, an improvement?

The most basic measure of coupling involves simply counting the number of other classes to which a given class has a linkage. If a CAR *is stored in* a GARAGE and *owned by* a PERSON, then assuming CAR, PERSON and GARAGE are three classes, CAR would have a coupling value of 2. By this measure, an understanding of which classes are most coupled within the system can be made.

* Corresponding author. Fax: +44-28-9036-6068.
E-mail addresses: fg.wilkie@ulst.ac.uk (F.G. Wilkie), barbara@cs.keele.ac.uk (B.A. Kitchenham).

PII: S0950-5849(01)00198-7

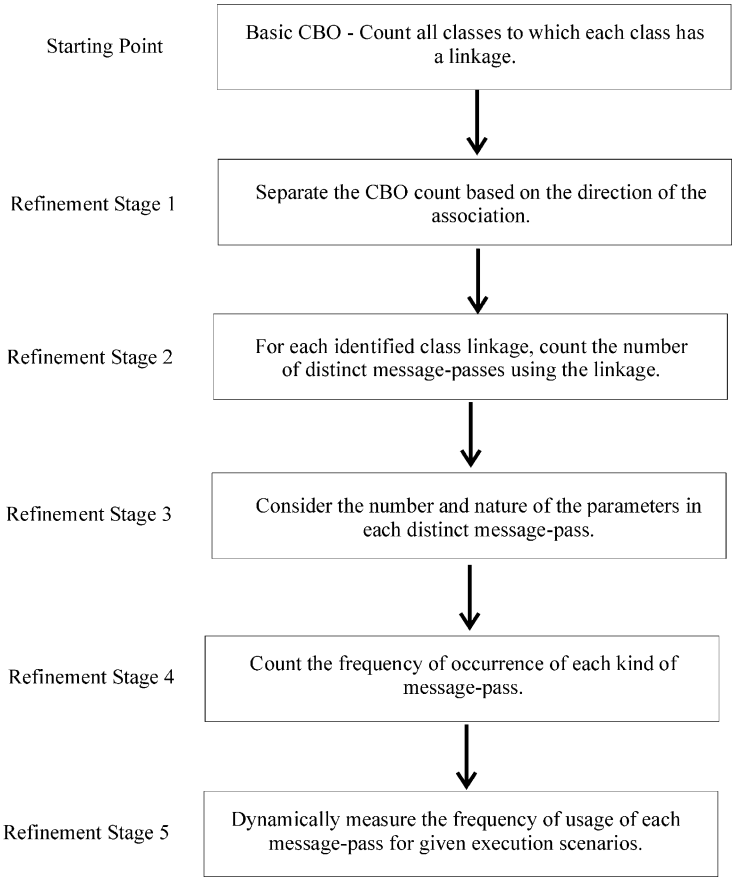

Fig. 1. Refinement stages to coupling measures.

There is a question concerning the direction of the association: should only bi-directional associations be counted or uni-directional associations as well? Henderson-Sellers et al. [7] discuss the evolution of the Chidamber and Kemerer [5,8] coupling metric, CBO. In Ref. [8], only bi-directional associations were counted. This constraint was removed in their later papers, so that any linkage from one class to another constitutes a coupling, regardless of there being a reciprocal reference in the coupled class. In this paper, we consider the direction of the association explicitly by measuring forward and backward associations separately.

Briand et al. [6] suggest that the strength of the coupling between two classes can be determined from two aspects: the frequency of connections between the classes and the types of connections between the classes. Following on from this, a more thorough treatment of coupling might include an analysis of the number of distinct messages passing from one class of objects to all those with which it is connected. The Message Passing Coupling (MPC) metric of Li and Henry [9] addresses this issue, as does the CCF metric proposed in Ref. [4] and used herein.

Continuing in this vein, we could also count the number of times that a given message-pass appears in each class. Further analysis in terms of association usage requires us to move from static to dynamic analysis — measuring the frequency of use of each message-pass for given execution scenarios. Several dynamic coupling metrics have been proposed [10] for this purpose. Such metrics are measuring object coupling rather than class level coupling [11] and are beyond the scope of this paper.

Returning to static coupling metrics, further refinements consider the number and nature (whether read-only or read-and-write) of parameters involved in the message-pass. In the case of C++, added complexity in the coupling occurs if 'friend' functions are invoked.

Fig. 1 summarises the progressively more detailed treatments of coupling that can be measured. Therefore, many levels of refinement and detail can be added to coupling metrics. The question is: *what level of refinement is required to make coupling metrics sufficiently useful predictors of software maintenance characteristics?* In the context of this paper, we have investigated the ripple effects in a commercial C++ application caused by changes to the application over a maintenance period of 2 1/2 years. In Ref. [12], a basic implementation of the Chidamber and Kemerer CBO metric employing only class linkage counting was used. The conclusions of that paper were that whilst the CBO measure identified the most change-prone classes, it did not identify those classes most vulnerable to ripple effect changes.

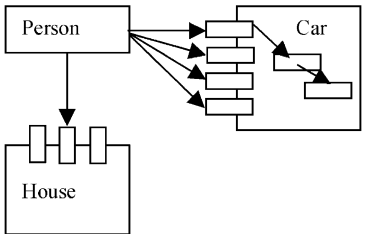

Fig. 2. Coupling Complexity (CCF and CCB).

This paper introduces refined versions of the CBO metric considering coupling strengths (in each direction) by counting the number of distinct message types that may use a class association, that is refinements corresponding to stages 1 and 2 in Fig. 1. By considering the number of distinct messages using a class coupling rather than simply counting class couplings will, we believe, lead to a more realistic appreciation of the coupling strengths involved between classes. For example, with reference to Fig. 2, it seems reasonable to assume that the coupling between Person and Car is greater than that between Person and House, since Person sends four distinct message types to Car but only one to House. Yet, using the CBO metric, both couplings contribute equally to the coupling strength of Person within this system of classes.

The hypothesis of the current paper is that these refinements will exhibit better diagnostic properties than the basic CBO, in predicting classes likely to be involved in ripple changes (that is, where a change introduced in one class has a ripple effect through a sub-set of the system of classes constituting an application).

Subsequent sections of this paper describe the extensions made to the basic CBO metric and the results of applying these new versions to the C++ application in an attempt to better predict the observed change ripples. Further predictive ability derived from the extensions to the coupling metrics that relates to software reuse are also described.

2. Extensions to the basic CBO metric

Two extensions have been made to the basic CBO metric to derive four new metrics: CBO(back), CCF(full), CCF(1) and CCB that are described in this section. CCF and CCB are fully specified in Ref. [4].

Briand et al. [6] developed a framework for categorising and comparing object oriented complexity metrics. This framework is mostly concerned with couplings resulting from interactions between classes. They identify three facets to such interactions:

1. Class–Attribute interaction, where one class is the type of an attribute of another class.
2. Class–Method interaction, where one class is the type of a parameter of a method in another class (or the return type of this method).
3. Method–Method interaction, where a method in one class either directly or indirectly invokes a method in another class.

Briand's framework also introduces the notion of a 'locus of impact' for an interaction, where a distinction is made between the client and server roles of the interacting classes.

The basic CBO metric used in Ref. [12] covers interaction facets (1) and (2) in Briand's framework. In this paper, we use two extensions. The first follows stage 1 in Fig. 1 where we have identified the need for a measure of backward coupling, CBO(back). This extension supports Briand's 'locus of impact' since it distinguishes between client and server roles. To make clear the distinction between CBO and CBO(back), we will refer to CBO as CBO(forward).

In a binary interaction between two classes *A* and *B*, a class, *A*, with either a Class–Attribute or Class–Method interaction to *B* is assumed to act as a client. This is measured by incrementing the CBO(forward) metric by one for class *A*. In the reverse direction, class *B* will have its CBO(back) metric incremented by one, indicating the assumption that class *B* fulfils the role of server. Looking at Fig. 2, the CBO(forward) value for *Person* is two and the CBO(back) value is zero, the CBO(forward) value for *Car* is zero and the CBO(back) is one.

Messaging interaction between classes (strictly speaking between corresponding objects of the class types) can only be assumed from the CBO metrics because neither actually measures Method–Method interaction — the third interaction facet of Briand's framework. This third facet is measured by our second refinement which is stage 2 of Fig. 1, building upon stage 1. The resulting metrics are called CCF and CCB. CCF stands for Coupling Complexity in the Forward direction of a class linkage and CCB stands for Coupling Complexity in the Backwards direction of a class linkage. The example below serves to illustrate CCF and CCB.

Consider Fig. 2. A small C++ source code listing for the scenario depicted in Fig. 2 is provided in Appendix A, along with comments relating to the calculation of CBO(forward), CBO(back), CCF(1), CCF(full) and CCB.

CCF measures the coupling in the forward direction from one class (say *Person*) to all other classes that *Person* has knowledge of (by virtue of some kind of reference). For each class that *Person* is coupled to, a count is made of all the distinct message-passes made from *Person* to that class — this is called CCF(1). The '1' in parenthesis indicates that only the called member function is included in the analysis. In addition, a count is made of the response set of member functions in the receiving class (eg. Car) for each received message — this metric is called CCF(full). The 'full' qualification indicates that the analysis includes a call trace within the receiving class of the inbound message. That is, a count is made of all member functions invoked within the receiving class as a result of the inbound message (the so-called response set).

We believe that CCF(1) is similar to the MPC metric [9]. From the example depicted in Fig. 2, for class *Person*, CCF(1) is five and CCF(full) is seven.

Fig. 2 shows two examples of uni-directional linkages since both Person–Car and Person–House involve interaction *from* Person *to* Car and House, respectively. There is no suggestion that either Car or House has explicit knowledge of Person. If they did, then bi-directional linkages would exist and distinct message passing from Car or House to Person could then take place. For example, the Car might want to notify the Person that their seat belt is undone. Such a message would be independent of any other interaction initiated by the Person and destined for the Car.

For simplicity, every linkage is assumed to be uni-directional. CCB measures the backward coupling as a result of an explicit forward coupling. For example, in Fig. 2, class *Car* has a CCF(full) of zero, but a CCB of four since there are four member functions referred to from other classes within the system (i.e. from class *Person* in this particular example). CCB takes account of the fact that while a uni-directional association has an explicit coupling in the forward direction, it also has an implicit coupling in the backward direction. In other words, if a Person sends a message 'Start Engine' to a Car, then there is an implied coupling from Car to Person even though there may be no explicit coupling from Car to Person. Explicit couplings enable messages to be sent, while implicit couplings do not. In this example, the act of starting the car may cause the Car to return a status result to the Person (success or fail). Such interaction is considered implicit in the context of this paper. This implicit coupling may be important when evaluating the impact of change on a system of classes. In Fig. 2, for class *House* CCB is one and for class *Person* CCB is zero.

Thus, the CCF metric is a measure of the extent to which a class requires services from other classes and the CCB metric is a measure of the extent to which a class provides services to other classes.

3. The analysed application

The analysed application has been reported elsewhere, for example Refs. [4,13], so that only brief details are provided here for clarity and completeness.

The analysed C++ application is a commercial multimedia conferencing system. One team of software engineers developed the first version of the application. The application was then handed over to a second team of developers for refinement and commercialisation. The overall application contains 114 classes, 1114 member functions, and approximately 25,000 lines of code.

The application's design architecture follows the traditional three layer model with a presentation layer, a business model layer and a network services layer.

Metrics analysis was performed manually on the first commercial release of the system to determine CBO(forward), CBO(back), CCF(full), CCF(1) and CCB. For all of these metrics, inherited associative and/or aggregate couplings were also included if used. This means that the CBO(forward) values differ from the CBO values reported in previous papers. Over the next 2 1/2 years, a detailed log of all significant changes[1] made to the application was kept. Changes came from three sources: (i) bug driven; (ii) customer driven and (iii) developer driven/redesign. All changes were given a unique ID that was noted in the change-log associated with each affected class (and each affected member function within that class). In this way, the rippling effects of changes on the system of interconnected classes could be traced. During the 2 1/2 year period of the study, 130 changes were applied to the software system and affected 26 of the 114 classes. These changes individually affected between 1 and 13 classes. In all, 44 changes caused ripples i.e. where two or more classes were affected.

Two version of the application were available for the extraction of coupling metrics. The first version was the version of the application first released to customers. The second version was the final released version of the application. During the maintenance of the application, some classes were created that did not appear in the first release of the application. In addition, some classes changed their name thus there is not a one-to-one relationship between the classes in the first released version of the application and classes in the final release.

In previous papers, [4,12,13], class coupling values were based on the first available measure. In most cases, the class values came from measurements taken of the first version of the C++ application, but in a few cases, the class values came from measures of the final version of the application. This happened because some new classes were introduced after the first version of the application but before the final version of the application. We now consider this approach to be incorrect. If we are interested in whether a class will be changed in the future, we should use the current class values and correlate them to subsequent changes. This procedure itself has inherent difficulties since the subsequent changes may themselves alter the values of the coupling measures. This is an implication of the dynamic nature of software applications and the fact that all static measures represent a snap-shot of an application at a specific point in time.

An initial identification of the difference between the first and final version of the application is that although change records identified 26 classes that were changed, only 24 of those classes appeared in the first version of the application.

All measures presented in this paper were made of the classes in the first version of the application. The classes in the final version of the application were simply used to determine what changes had actually taken place during the intervening 2 1/2 year period of maintenance. The

[1] Minor changes such as removal or inclusion of debug code may not have been recorded in all cases.

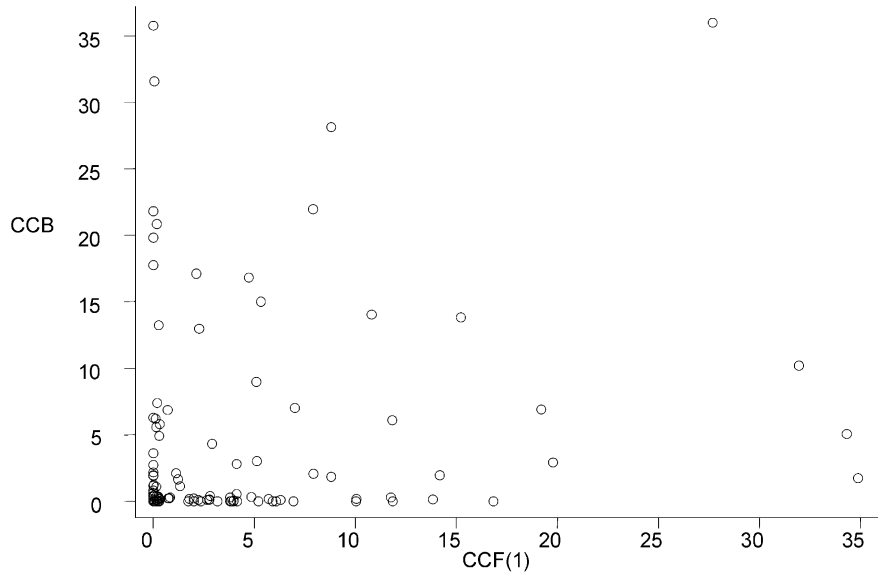

Fig. 3. Scatterplot of CCB versus CCF(1) (jitter applied to show duplicate points).

metrics from the first version of the application were correlated with information from the change logs in the final version of the application in order to determine their predictive capabilities.

All statistical analyses were performed using Intercooled STATA for Windows 95 [14].

4. Results

The data set has a high proportion of public member functions. As a consequence, the correlation between CCF(full) and CCF(1) was very high, at 0.942. We used Kendall's tau b to assess the correlation because the metrics are not normally distributed. Kendall's tau b is a variant of Kendall's tau that adjusts for multiple data points having the same value. The difference between CCF(full) and CCF(1) is derived from the response set for each public member function. When a high proportion of the member functions in a class are public, the response set for each public function tends to 0, i.e. many public member functions do not make any calls to other member functions within that class. Since CCF(1) is a measure of the number of called public member functions, CCF(full) approximates to CCF(1) in this application. The following results therefore use only CCF(1).

Fig. 3 shows a scatter plot of CCF(1) versus CCB for our class set. The scatter plot shows three broad categories into which the C++ classes fall. These are: (a) classes for which CCF(1) = 0 and CCB > 0; (b) classes for which CCB = 0 and CCF(1) > 0 and (c) classes for which both CCF(1) and CCB have significant values. Characteristics of the classes falling into each of these categories were investigated and are discussed in the following sections.

Table 1
Analysis of classes with CCF(1) = 0 and CCB > 5

Class ID	Brief description	CBO(f)	CBO(b)	CCB	Architectural layer	DIT[a]	NOC[b]	Reuse potential[c]
Class 1	Handles connections	0	4	18	Network	0	1	High
Class 2	Session info associated with a connection	0	12	22	Network	0	0	High
Class 3	Preparation for data marshalling	1	17	32	Network	0	0	High
Class 4	Data marshalling	0	5	6	Network	0	2	High
Class 5	Part of wire protocol	1	3	21	Network	0	0	High
Class 6	Part of session setup	2	8	13	Network	0	2	High
Class 7	Info about a user	0	4	6	Model	0	0	High
Class 8	Generalised shape	0	13	36	Presentation	0	7	High
Class 9	Info about session users	0	5	36	Model	0	0	High
Class 10	–	2	3	6	Network	0	0	Medium
Class 11	Part of network protocol	0	1	6	Network	0	3	High

[a] DIT = Depth in the Inheritance Tree.
[b] NOC = Number of direct Children.
[c] Opinion of developer.

Table 2
Correlations between coupling metrics and number of changes per class for all 114 classes

Coupling metric	Kendall's tau b	Significance
CBO(forward)	0.29	$p < 0.01$
CBO(back)	0.12	Not significant
CCF(1)	0.34	$p < 0.01$
CCB	0.12	Not significant

4.1. Classes with CCF(1) = 0

These classes have no internally sourced message-passing activity. This suggests that such classes may be service providers, simply reacting to calls from other sources within the application.

We manually inspected the source code belonging to each of the 11 classes with CCF(1) values of zero and CCB values of six or more. (Our choice of classes with a CCB value >5 was based on a visual examination of Fig. 3) In all there were 27 classes with CCB > 5, i.e. 24% of the total number of classes, but only 11 of those classes also had CCF(1) = 0. We used the inspection as an opportunity to develop a detailed understanding of the purpose of each of the classes in question. The Chief Technical Architect on the project was interviewed to determine his opinion of the reusability of each of these 11 classes. Table 1 shows the results obtained.

All 11 classes were potentially reusable in the opinion of the architect. Some were reusable in the context of that business domain (indicated by a 'medium' rating) while others were potentially reusable in other domain contexts (indicated by a 'high' rating). It is interesting to note that five of the classes are located at the root of inheritance hierarchies (i.e. have DIT = 0 and NOC > 0) where reusable classes might be expected.

Four of the classes have non-zero values for the CBO(forward) metric. Therefore, the CBO(forward) metric on its own may not be the most useful predictor of reusable classes. Any form of coupling limits reusability. However, some classes appear to store and pass object references around the application, without actually using the references for the purposes of interaction (i.e. message passing). For example, a class may store a reference to an object of another class and simply return this reference to a caller.

Such classes are obviously not devoid of application context, but they often fulfil roles that can be abstracted into generic and highly reusable units.

The other seven classes have a CBO(forward) metric of zero. These classes may present the most reuse potential since they contain little if any application context. Their roles are widely applicable.

Eight of the 11 classes are located in the Network layer which is normally subservient and could therefore be expected to yield more reusable classes.

4.2. Classes with CCB = 0

A metric value of zero for CCB suggests classes that are primarily concerned with consuming services. These classes are expected to be highly context dependent and therefore to exhibit less reuse potential. We found that of the 33 classes for which CCB = 0 and CCF(1) is non-zero, 17 belong to aspects of the presentation layer. This is consistent with modern design where the underlying model layer classes are often decoupled from the presentation layer, with presentation objects requesting information from the model layer when needed.

4.3. Classes with significant values for both CCB and CCF(1)

There is a group of seven classes for which both the CCB and CCF(1) metrics have significant values. These classes are tightly coupled within the application. They provide services but also demand services from other classes. They are central to the architectural layer within which they reside and as such need to be very carefully tested. In our data set, five of the classes come from the Business Model layer and two from the Network Services layer. When we showed this result to the developer, he confirmed that the five business model classes were indeed five of only six classes that are considered core to the overall application. The metrics have therefore been able to accurately identify the broad features of the application under study.

4.4. The relationship between coupling metrics and changes

The correlation between the total number of changes (from all sources) associated with a class and the coupling metrics is shown in Table 2. We used Kendall's tau b as a non-parametric measure of correlation because the metrics are not normally distributed. Inspection of Table 2 shows that CCF(1) has a slightly larger correlation with number of changes per class than CBO(forward). However, if our original hypotheses were valid we would expect CCF(1) to have a significantly larger correlation with number of changes that CBO(forward) and this is not the case. We also note that backward coupling is not associated with change-proneness.[2] This is similar to the results for conventional procedural systems where fan-out is correlated with changes and errors, but fan-in is not [15].

For both the forward coupling metrics, we also used logarithmic regression to identify whether or not a class will be changed (irrespective of the number of changes). We constructed a dependent variable (*Changed*) that took the value zero if a class had not been changed between the first and final versions of the application and one if it was changed between the two versions. We then performed two separate logistic regression analyses using each of the forward

[2] Change-proneness means the likelihood that the class will be involved in change during the maintenance phase.

Table 3
Logistic regression results using a dependent variable indicating whether or not a class was changed after release (90 unchanged classes and 24 changed classes)

Independent variable	Odds ratio	Standard error	Significance	95% Confidence limits	
				Lower	Upper
CBO(f)	1.5903	0.2066	< 0.01	1.2328	2.0516
CCF(1)	1.1862	0.0537	< 0.01	1.08559	1.2962

coupling measures in turn as the single independent variable. The results of the two logistic regressions are shown in Table 3 (for logarithmic regression, an independent variable with an odds ratio significantly greater than one has a significant effect on the dependent variable).

Table 3 shows that the forward coupling metrics are predictors of change-prone classes but indicates that CBO(forward), with a much larger odds ratio, is better than the more refined CCF(1). El-Emam et al. [16,17] note that size measures are often correlated with coupling measures. Thus, if there is an association between size and defect rate or change rate, then we may observe a correlation between a coupling measure and defect rate or change rate as a side-effect of the relationship between coupling and size. To check the effect of size, we added number of functions per class as a second independent variable to each logistic regression. The results are shown in Table 4.

The results imply that, both CBO(forward) and CCF(1) are significant predictors of change-proneness after allowing for the effect of size. Again the CBO(forward) metric appears to be a better predictor than CCF(1). This result contradicts our original hypothesis.

An explanation for our results is that CCF represents refinement stage 2 in Fig. 1. It may be that we need to consider refinement stages 4 or 5 to build a more realistic understanding of the actual couplings. For example, consider again the small scenario presented in Fig. 2. There are four distinct message passes from Person to Car and only one from Person to House. By our refined metric (CCF), the coupling between Person and Car would therefore be greater than that between Person and House. However, if there are 10 sites in the C++ source code where the single message pass is made between Person and House, but only one site for each of the four message passes between Person and Car, then there may be a greater probability of having a messaging interaction between Person-and-House than Person-and-Car. This would lead to a higher coupling between Person-and-House than Person-and-Car (based on refinement 4). By introducing refinement stage 5 which involves dynamic analysis, we might discover that the dynamic coupling between Person-and-Car is about the same as that between Person-and-House — a result that would (coincidentally) be exactly the prediction of the basic CBO metric. Hence, the picture is quite complicated and requires further research using more refinements to the coupling metrics in order to disambiguate our results. All that we can say from our results is that the refinement stage 2 coupling metric (CCF) appears not to provide any additional change-proneness predictive capabilities beyond those of the basic CBO metric for our dataset.

Considering only those 24 classes that were changed, the association between functions per class, CBO(forward), CCF(1) and number of changes is shown in Table 5. None of the correlations was significantly different from zero. This indicates that although the metrics are predictors of change-proneness, they do not predict the frequency of change.

4.5. Predicting change ripples using CCF(1) and CBO(forward)

One of our goals in developing the CCF(1) metric was to better identify classes likely to be involved in ripple changes (i.e. changes that involve more than a single class), so it is important to see how CCF(1) compares with CBO(forward) with regard to predicting change ripples. Running the logistic regression for all classes that had zero ripple changes compared with one or more ripple changes, the results were similar to the results of the logistic regression based on simple change-proneness (see Table 6). In this logistic

Table 4
Logistic regression results using a dependent variable indicating whether or not a class was changed after release with both size and coupling as independent variables (90 unchanged classes and 24 changed classes)

Independent variables	Odds ratio	Standard error	Significance	95% Confidence limits	
				Lower	Upper
CBO(f)	1.4808	0.2148	< 0.01	1.1144	1.9678
Functions	1.1145	0.0345	< 0.01	1.0489	1.1842
CCF(1)	1.1378	0.0539	< 0.01	1.0369	1.2485
Functions	1.0971	0.0358	< 0.01	1.0291	1.1695

Table 5
Correlation between changes and size and coupling measures (Kendall's tau-b) for 24 classes that were changed after first release

Metric	Correlation with number of changes (Kendall's tau b)
Functions per class	−0.13 (not significant)
CBO(forward)	0.20 (not significant)
CCF(1)	0.00 (not significant)

regression, 22 classes had ripple changes and 92 did not, i.e. two classes changed category because none of their changes involved ripples. Table 6 confirms that CBO(forward) is a better predictor of ripple-change proneness[3] than CCF(1). However, since only two of the classes changed category when we considered ripple changes it is not surprising that the results are similar to the results for simple changes.

The reason for the increase in the CCB(forward) odds ratio and the decrease in the CCF(1) odds ratio compared with Table 4 can be explained by considering what a logistic regression actually does. A logistic regression identifies a cut-off value of each input variable such that data points with values below the cut-off value are assigned to the one group and classes with values above (or equal to) the cut-off value are assigned to the other group. The reason CCF(1) appears to be a worse predictor of ripple changes than it is of simple changes is because one of the classes that changed category had a very large CCF(1) value (i.e. 34). This class is therefore mis-classified for the ripple-change proneness model whereas it was correctly classified for the changes model. In contrast, the relative improvement in CBO(forward) is because both of the classes that changed classification had relatively low CBO(forward) values, i.e. 2 and 4. This means they were both mis-classified in the change-proneness logistic model but correctly classified in the ripple changes logistic.

In addition to logistic regression, we investigated the correlations between the number of ripple changes and coupling and size measures for classes that were changed. Table 7 shows the association between functions per class, CBO(forward) and CCF(1) and the number of ripple changes (i.e. changes that involved more than one class). These show no significant correlations between any of the metrics and the number of ripple changes.

We also investigated the relationship between the coupling measures and the proportion of changes that were ripple changes, for the 14 classes for which the number of changes was four or more. The correlations are shown in Table 8. Again, none of the correlations was significantly different from zero.

Thus, we conclude that there is evidence that CBO(forward) and CCF(1) can identify change prone classes and ripple-change prone classes, but neither metric can predict the extent of simple changes or ripple changes. In contradiction to our original hypothesis, CBO(forward) is a better predictor of change-prone classes than CCF(1).

[3] Ripple-change-proneness means the likelihood that the class will be involved in changes that affect more than one class.

5. Study limitations

This study has five major limitations that need to be considered when interpreting its results.

1. This is a study of a single C++ application, so it is equivalent to a case study. As such, the scientific inferences that can be drawn from it are limited to identifying examples or counter examples. That is, we can demonstrate that a phenomenon can be observed or cannot be observed. However, we cannot make any strong claims about the generality of any of the observed phenomena, and, in particular, we do not claim that any of the models we built have any validity outside the context of the specific data set on which they were developed.
2. We have treated all changes as equivalent whether they arose from requirements changes or defect corrections. We justified this approach in previous studies ([4,12]). However, it makes the results somewhat more difficult to interpret.
3. We use a subjective assessment of the reuse potential of the classes provided by a single person. However, the person we discussed our findings with was the chief technical architect on the project. He had overall technical control of the group and was the only person with a 'global' view of the entire system. Therefore, he was well placed to answer our questions.
4. We do not take inheritance fully into account in our analysis. We take some account of inheritance in the construction of our metrics, that is, if an association was inherited into a sub-class, then that association contributes to the coupling metric values. However, beyond that, the study does not investigate inheritance.
5. This is a regression and correlation study. As such, we can confirm associations among class metrics but we cannot make any claims about causality. Thus, we cannot claim that altering a class to reduce its coupling metric values would decrease its likelihood of being changed. Such claims could only be made after extensive controlled experiments into alternative design approaches. Our conclusions are, therefore, restricted to suggesting how software engineers can use the actual coupling metrics to assist development activities, not how they could alter the metric values.

6. Conclusions and future work

Classes tend to fall into three broad categories: (a) service requesters; (b) core application logic and (c) service providers. Classes in category (b) may require a disproportionate

Table 6
Logistic regression results using a dependent variable indicating whether or not a class was involved in a ripple change after release with both size and coupling as independent variables (92 unchanged classes and 22 ripple-changed classes)

Independent variables	Odds ratio	Standard error	Significance	95% Confidence limits	
				Lower	Upper
CBO(f)	1.6122	0.2395	< 0.01	1.2050	2.1570
Functions	1.0796	0.0305	< 0.01	1.0214	1.1411
CCF(1)	1.1027	0.0433	< 0.01	1.0210	1.1909
Functions	1.0651	0.0317	< 0.05	1.0047	1.1292

amount of time and effort devoted to their testing and proper integration into the application.

The results show that the combination of CCF(1) and CCB can provide additional insight into the underlying classes and their suitability for reuse. Some classes within an application act as repositories for object references without themselves ever using the references to pass messages. Such classes fulfil roles that may be abstracted into generic and hence reusable classes. Furthermore, we believe the CCF(1) and CCB metrics together should help identify classes that require different testing strategies. Classes with zero CCF(1) and high CCB values need to be extremely reliable since they will provide services to many different classes in the application. They can however, be tested in isolation from the rest of the system as long as the testing activity includes black-box functional tests. Classes with high values of CCF(1) and CCB will require extensive integration testing because such classes can interact with other classes in very complex ways. Classes with zero CCB and high values of CCF(1) require services from other classes, they can be tested using a bottom up strategy by waiting until the service providing classes have been tested, or in a top down fashion by the provision of stubs.

Interestingly, the largest proportion of classes that suffered change over the maintenance period come from the service requestor category and not the core application logic. This suggests the need to be particularly careful in the design of the driver classes. Correctly identifying the driver classes can aid developers when performing impact analysis.

Both CBO(forward) and CCF(1) are predictors of change proneness. However, contrary to our expectation, the more refined extension of CBO i.e. CCF(1) was not as good a predictor as CBO(forward). Furthermore, neither metric was able to identify the extent of change-proneness. This suggests that detailed refinements of the CBO metric may be unnecessary unless, like CCF(1) and CCB, they offer useful insights into the architecture of an object-oriented system, such as the opportunity to identify reusable components, or components that require additional testing effort, or classes with potential code inconsistencies or redundancy.

Lindvall and Sandahl [18] have carried out studies on the ways in which software engineers perform impact analysis for change requests during the maintenance cycle of software projects. They observed that experienced software engineers prefer to rely on implicit models held in their minds rather than use the documentation and models accompanying a system, when performing impact analysis. They also concluded that "there is a tendency to underestimate the number of changed classes". The use of coupling metric profiles for software systems may help ensure that developers have a more comprehensive grasp of the extent of the interrelationships present in a system of classes.

Our future research effort will concentrate on investigating whether coupling metrics can provide practical assistance to the developers of object-oriented systems. In particular, we believe it is important to investigate the way in which object-oriented systems evolve towards a final solution. In the scenario of rapid system evolution, we need to understand the ways in which object-oriented metrics can assist designers and managers in understanding the current status of the system, and to be able to assess whether it is converging towards a stable, reliable state or not.

Table 7
Correlation between ripple changes and coupling measures (Kendall's tau b) for 24 classes that were changed after first release

Metric	Correlation with number of ripple changes (Kendall's tau b)
Functions per class	−0.19 (not significant)
CBO(forward)	0.25 (not significant)
CCF(1)	−0.02 (not significant)

Table 8
Correlation between the proportion of ripple changes per class and coupling measures (Kendall's tau b) for 14 classes that were changed more than three times after first release

Metric	Correlation with the proportion of ripple changes (Kendall's tau b)
Functions per class	−0.19 (not significant)
CBO(forward)	−0.06 (not significant)
CCF(1)	−0.10 (not significant)

We also intend to explore ways of extending the coupling metrics through refinement steps 4 and 5 which will provide additional insight into the coupling forces within object oriented software. This may also help in further explaining some of the results obtained and presented in this paper.

To summarise, we believe that the following professional software development staff can benefit from using the coupling metrics profiles discussed in this paper.

1. Reuse Engineers can use the metrics to rapidly identify reuse candidates that they may wish to consider for inclusion in reusable component libraries.
2. Software Test Engineers can use the metrics to identify the type of testing strategy that will best suit the various classes involved in the application.
3. Software Maintenance staff can use the metrics to help with impact analysis and to identify those parts of the code with which they should become most familiar because they exhibit the greatest likelihood of change over time.

Acknowledgements

The authors wish to thank CCC Technology Ltd for their assistance during this project.

The dataset used in this paper can be made available if required for independent audit under conditions of commercial confidentiality.

Appendix A. C++ source code listing for scenario of Fig. 2

There are several ways that linkages from Person to Car and House could be created. These are: by embedded pointers; aggregation or as parameters in public member functions of the class Person. In this example, the Person class has an embedded pointer to a Car and takes a pointer to a house as a parameter in a member function call. The example is for illustrative purposes only and is not intended to constitute a complete application.

```
Class Person
  {
  public:
   void purchase_home(House *aHouse);   //linkage to house passed
                                        //as a parameter .
                                        //increments CBO value

  private:
   Car *aCar;   //embedded pointer creating linkage to Car class-
                //increments CBO value.

  };
void Person::purchase_home(House *aHouse)
  {
   aHouse → open_door();   //Message pass to house, increments
                           //CCF( full) and CCF(1)

  aCar → start();   //Message pass to car, inc CCF(full) and CCF(1)
  aCar → select_gear(1);   //Message pass to car, inc CCF(full) and CCF(1)
  aCar → accelerate(10);   //Message pass to car, inc CCF(full) and CCF(1)
  aCar → brake(4);   //Message pass to car, inc CCF(full) and CCF(1)
  };
```

CBO(forward) = 2
CBO(back) >= 1 (there must be a link somewhere to call 'purchase_home')
CCF(full) = 7 (requires an appreciation of called classes, i.e. Car and House)

CCF(1) = 5
CCB >= 1 (since function 'purchase_home' is called from somewhere!)

```
Class House
  {
  public:
    void open_door(void) {door_status = TRUE;};
    void close_door(void) {door_status = FALSE;};
    void open_window(void){window_status = TRUE;};
  private:
    bool door_status, window_status;
  };
```

CBO(forward) = 0 (since House has no linkages to other classes)
CBO(back) = 1 (open door is called by Person)
CCF(full) = 0
CCF(1) = 0
CCB = 1

```
Class Car
  {
  public:
    void start(void){status = TRUE;};
    void select_gear(int gr) {gear = gr;};
    void accelerate(int);
    void stop(void){speed = 0;};
  private:
    long speed;
    int gear;
    bool status;
    void calculate_speed(int);
    int get_time(void);
  };
void Car::accelerate(int degree)
  {
  calculate_speed(degree); // inc CCF(full) in caller but not CCF(1)
  };
void Car::calculate_speed(int accel)
  {
   speed = speed + accel*get_time(); //increment CCF( full) in caller
                                     //but not CCF(1)

  };
int Car::get_time(void)
  {
  int time;
  cout << "How long do you wish to apply this condition for?"
  cin >> time;
  return(time);
  };
```

CBO(forward) = 0
CBO(back) = 1 (class Person links to Car)
CCF(full) = 0
CCF(1) = 0
CCB = 4 (since one class i.e. Person invokes 4 member functions in Car)

References

[1] M.J. Fyson, C. Boldyreff, Using application understanding to support impact analysis, Journal of Software Maintenance: Research and Practice 10 (1998) 93–110.
[2] R.S. Arnold, S.A. Bohner, Impact analysis — towards a framework for comparison, International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1993 pp. 292–301.
[3] M. Lindvall, Measurement of change: stable and change-prone constructs in a commercial C++ system, Proceedings of the Sixth International Software Metrics Symposium, November 4–6, IEEE Computer Society Press, Los Alamitos, CA, 1999 pp. 40–49.
[4] F.G. Wilkie, B. Hylands, Measuring complexity in C++ application software, Software Practice and Experience 28 (5) (1998) 513–546.
[5] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[6] L.C. Briand, J.W. Daly, J.A. Wüst, Unified Framework for coupling measurement in object-oriented systems, Fraunhofer Institute for Experimantal Software Engineering, Kaiserslautern, Germany, Technical Report number ISERN-96-14, 1996.
[7] B. Henderson-Sellers, L.L. Constantine, I.M. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design), Object Oriented Systems 3 (1996) 143–158.
[8] S.R. Chidamber, C.F. Kemerer, Towards a metrics suite for object-oriented design, Proceedings of OOPSLA'91, Sigplan Notices 26 (11) (1991) 197–211.
[9] W. Li, S. Henry, Object oriented metrics that predict maintainability, Journal of Systems and Software 23 (1993) 111–122.
[10] S.M. Yacoub, H.H. Ammar, T. Robinson, Dynamic Metrics for Object Oriented Designs, Proceedings of the Sixth International Software Metrics Symposium, Florida, November 4–6, IEEE Computer Society Press, Los Alamitos, CA, 1999 pp. 50–61, ISBN 0-7695-0405-5.
[11] M. Hitz, B. Montazeri, Measuring coupling and cohesion in object-oriented systems, Proceedings of the International Symposium on Applied Corporate Computing, Monterrey, Mexico, October 1995.
[12] F.G. Wilkie, B.A. Kitchenham, Coupling measures and change ripples in C++ application software, Journal of Systems and Software 52 (2000) 157–164.
[13] F.G. Wilkie, I. O'Neill, S. Tripathy, C. McCauley, Reuse management of complex distributed computing components, Object World Conference, Boston, MA 1995 pp. 303–320.
[14] STATA Corporation. Intercooled STATA 5.0 for Windows 95.
[15] B.A. Kitchenham, L.M. Pickard, S.J. Linkman, An evaluation of some design metrics, Software Engineering Journal 5 (1) (1990) 50–58.
[16] K. El-Emam, S. Beniarbi, N. Goel, The confounding effect of class size on the validity of object-oriented metrics, National Research Council Canada, Report ERB-1062, September 1999.
[17] K. El-Emam, S. Beniarbi, N. Goel, S. Rai, A validation of object-oriented metrics, National Research Council Canada Report, ERB-1063, October 1999.
[18] M. Lindvall, K. Sandahl, Traceability aspects of impact analysis in object-oriented systems, Journal of Software Maintenance: Research and Practice 10 (1998) 37–57.